Domain Morphisms: A New Construct for Parallel Programming and Formalizing Program Optimization

Marina Chen and Young-il Choo

YALEU/DCS/TR-817
August 1990

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Yale University
Department of Computer Science

Domain Morphisms: A New Construct for Parallel Programming and Formalizing Program Optimization

Marina Chen and Young-il Choo

YALEU/DCS/TR-817
August 1990

This work has been supported in part by the Office of Naval Research under Contract N00014-89-J-1906, N00014-90-J-1987.

Domain Morphisms: A New Construct for Parallel Programming and Formalizing Program Optimization

Marina Chen Young-il Choo

Department of Computer Science
Yale University
New Haven, CT 06520
chen-marina@yale.edu choo@yale.edu

August 1990

Abstract

This paper addresses the issue of how to make the task of writing correct and efficient parallel programs easier. *Domain morphism* is a new construct for specifying parallel-program optimization at a high level, and enables an equational program transformation whereby the optimized program containing implementation details is derived mechanically. In addition, we introduce the constructs *index domain*, *data field*, *communication form*, and *hyper-surfaces* for facilitating parallel programming and generating efficient target parallel programs with explicit synchronization and communication. We describe an equational theory and new program transformation procedures motivated by these new constructs. A programming example illustrating the use of these constructs and the program transformation steps is given.

Contents

1 Introduction

Writing correct programs is not easy; making programs run efficiently requires further complex process of matching the logical structure of the program to the underlying physical structure of the machine. With the arrival of large-scale multi-processor computers, we face the challenge of parallel programming where the allocation of processors and the cost of inter-processor communication now enter into the question of efficiency.

This paper addresses the issue of how to make the task of writing correct and efficient parallel programs easier. Our approach to programming starts with a high-level problem specification, through a sequence of optimizations tuned for particular parallel machines, leading to the generation of efficient target code with explicit synchronization and communication. The key new concept of *domain morphism* specifies the transformation from one program to the next in the sequence. It plays a dual role, first, as a construct for specifying an optimization strategy, and second, in enabling an equational program transformation whereby the optimized program can be derived mechanically.

Domain morphisms capture, as mathematical objects, the essence of existing loop transformations in vectorizing and parallelizing compilers, as well as new techniques of data layout and efficient inter-processor communication for distributed and shared-memory machines with memory hierarchy.

Formalizing the optimizations a parallelizing compiler must do requires the notion of domain morphism, which in turn requires an equational theory of programs. We have found the traditional program transformations inadequate in that operators such as folding and unfolding [?, ?, ?]) are applied only to expressions, whose extensions remain unchanged. The equational theory allows us to derive a new program which is equivalent, in a precise way defined by the domain morphism, but which is extensionally different from the original in that it is defined over the new domain.

The rest of the paper is organized as follows. In Section ??, we introduce the domain morphism and other constructs for specifying parallel program transformations. Section ?? introduces the equational theory for program transformation. The next section presents in full detail an example of domain morphism induced program transformation on a program for solving Laplace's equation using successive over-relaxation (SOR).

2 New Constructs

In the following, we introduce a few programming constructs which are important for parallel programming, where of central importance is the notion of *domain morphism*. *Index domains* are abstractions of the "shapes" of composite data structures, which in most current programming languages are not first-class objects. *Data fields* generalize the notion of distributed data structures, unifying the conventional notions of arrays and functions. *Communication forms* defined over an index domain are means of specifying the data dependencies, and the inverse of a communication form (defined precisely later) provides useful directives for optimizing inter-processor communication. *Hyper-surfaces* are for specifying the boundaries of index domains. Finally, *domain morphisms* describe the reshaping of index domains aimed at optimizing data or control structures for efficiency reasons as well as

the necessary mapping from the logical structure of the problem to the physical domain of machine and sequencing in time.

Notation A function f from domain A to codomain B is denoted $f : A \to B$, and is defined using function-abstraction: $f = \mathsf{fn}(x){:}A\{\tau[x]\}$. The conditional construct is represented in a two-dimensional form $\left\{\begin{matrix} b_1 \to h_1 \\ b_2 \to h_2 \end{matrix}\right\}$ and its meaning is h_i if b_i is true for one i, and undefined otherwise.

2.1 Index Domains

An *index domain* D consists of a set of elements (called indices), a set of functions from D to D (called communication operators), a set of predicates, and communication cost associated with each communication operator.

In essence, an index domain is a data type with communication cost associated with each function or operator. The reason for making the distinction is that index domains will usually be finite and they are used in defining distributed data structures (as functions over some index domain), rather than their elements being used as values. For example, rectangular arrays can be considered to be functions over an index domain consisting of a set of ordered pairs on a rectangular grid. Also, the elements of an index domain can be interpreted as locations in a logical or real space and time over which the parallel computation is defined. So we classify index domains into certain *kinds* (second order types) according to how they are to be interpreted (for example, as time or space coordinates).

Basic Index Domains We now give examples of basic classes of index domains. The most often used are the interval and the hypercube domains:

An *interval domain*, denoted $\mathsf{interval}(m, n)$, where m and n are integers and $m \leq n$, is an index domain whose elements are the set of integers $\{ m, m+1, m+2, \ldots, n \}$ with the usual integer functions and predicates. The communication operators are $\mathsf{prev} : i \mapsto i - 1$ and $\mathsf{next} : i \mapsto i + 1$, with communication cost 1. The operators lb and ub return the lower bound (m) and the upper bound (n) of the interval domain respectively. When $m \geq n$, we define the index domain to be the same except that prev and next have reversed meaning.

A *hypercube domain* of dimension n, denoted $\mathsf{hcube}(n)$, is an index domain with 2^n elements of the form $(x_0, \ldots, x_{n-1})$, where each x_i is either left or right (or just 0 or 1), and communication operators $\mathsf{hcnb}(j, k)$, for j an element of the data type and $0 \leq k \leq n - 1$, which maps the element j to its neighbor along the kth dimension, each with unit communication cost. Predicates are $\mathsf{left?}(k)$ and $\mathsf{right?}(k)$, for testing whether an element is on the left or right half of the kth dimension.

Index Domain Constructors Given index domains D and E, we can construct their product $(D \times E)$, disjoint union (coproduct) $(D + E)$, and function space $D \to E$, in the usual way.

2.2 Kinds of Index Domains

As discussed above, it is useful to indicate how an index domain is to be interpreted by "typing" index domains with "kinds" (second-order types). Analogous to the first-order typing, $D[\mathcal{K}]$ will mean that the index domain D is of kind $\mathcal{K}$. The following are the kinds we have found useful for compiler optimizations:

Universal ($\mathcal{U}$): the kind of any index domain as well as any other data type, such as the integers and the reals.

Temporal ($\mathcal{T}$): the kind of index domain representing time coordinates (subkind of $\mathcal{U}$).

Spatial ($\mathcal{S}$): the kind of index domain representing space coordinates (subkind of $\mathcal{U}$).

Processor ($\mathcal{P}$): the kind of index domain representing processor coordinates (subkind of $\mathcal{S}$).

Memory ($\mathcal{M}$): the kind of index domain representing memory locations within a processor (subkind of $\mathcal{S}$). Memory hierarchy can be introduced with other kinds $\mathcal{M}_i$ which are subkinds of $\mathcal{M}$, where i ranges over the levels of the memory hierarchy.

2.3 Index Domain Morphisms

Index domain morphisms formalize the notion of transforming one index domain into another, with the kinds of the domains indicating the change of interpretations.

Definition Let D and E be two index domains. An *index domain morphism* is a function g from D to E such that for all elements x and y in D, if there exists a composition τ of communication operators $\tau_1 \circ \tau_2 \circ \cdots \tau_k$ over D such that $y = \tau(x)$, then there is a composition τ' of communication operators $\tau'_1 \circ \tau'_2 \circ \cdots \tau'_k$ over E such that $g(y) = \tau'(g(x))$.

The extra constraint is to ensure that if there is a path in the first domain from x to y, then there is a path from $g(x)$ to $g(y)$.

Reshape Morphisms For any index domain morphism $g : D \rightarrow E$, a *left inverse*, if it exists, is an index domain morphism $h : E \rightarrow D$ such that $h \circ g = 1_D$, the identity morphism over D. If g is also a left inverse of h (i.e., $g \circ h = 1_E$), then h is called the *inverse* of g and is denoted by g^{-1}. A morphism that has an inverse is commonly known as an isomorphism but here will be called a *reshape morphism* to emphasize the idea of "reshaping" one index domain into another.

To require the existence of the left inverse implies that a reshape morphism must be bijective. However, we can easily derive a reshape morphism from an injective domain morphism by restricting the codomain as follows: Given an injective domain morphism $g' : D \rightarrow E$, the image of D under g', denoted $\mathbf{image}(D, g')$, is an index domain whose elements are the image of the elements of D and whose communication operators are those of E. Clearly, $g : D \rightarrow \mathbf{image}(D, g')$, derived from g', now has a well-defined left inverse.

Here are examples of some useful reshape morphisms:

- An *affine morphism* is a reshape morphism which is an affine function from one product of intervals to another. Affine morphisms unify all types of loop transformations (interchange, permutation, skewing) [?, ?, ?, ?, ?], and those for deriving systolic algorithms [?, ?, ?, ?]. For example, if $D_1 = \mathsf{interval}(0,3)$ and $D_2 = \mathsf{interval}(0,6)$, then $g = \mathsf{fn}(i,j) : D_1 \times D_2\{(j,i) : D_2 \times D_1\}$ is an affine morphism which effectively performs a loop interchange.

  Another example illustrates a slightly more interesting codomain E of the morphism g by taking the image of a function g'.

$$\begin{aligned} D_0 &= \mathsf{interval}(0,3) \qquad D = D_0 \times D_0 \\ E &= \mathsf{image}(D, g') \text{ where}\{\ g' = \mathsf{fn}(i,j){:}D\{(i-j, i+j)\}\ \} \\ g &= \mathsf{fn}(i,j){:}D\{(i-j, i+j){:}E\} \qquad g^{-1} = \mathsf{fn}(i,j){:}E\{((i+j)/2, (j-i)/2){:}D\} \end{aligned}$$

  Whenever it is legal to apply this affine morphism to a 2-level nested loop structure—consistent with the data dependencies in the loop body [?, ?, ?, ?, ?, ?])— a similar structure, but "skewed" from the original, is generated. The most common case is when elements of the inner loop can be executed in parallel, but where only half of the elements are active in each iteration of the outer loop. In this example, the index domain E has holes, and so guards in the loops must test whether $i+j$ and $i-j$ are even, since only these points correspond to the integral points in D

  In Section ??, we will show in detail how such a transformed loop is derived algebraically.

- A *uniform partition* of a domain D is a reshape morphism $g : D \to D_1 \times D_2$, with the property that the codomain has a greater number of component domains (i.e., is of higher dimensionality).

  For example, If $D = \mathsf{interval}(0,11)$, $D_1 = \mathsf{interval}(0,3)$, and $D_2 = \mathsf{interval}(0,2)$, then

$$g = \mathsf{fn}\, i : D[\mathcal{S}]\{(i/3, i \bmod 3) : D_1[\mathcal{P}] \times D_2[\mathcal{M}]\}$$

  is a uniform partition that distributes the elements of an index domain of the spatial kind as follows: the 12 elements are partitioned into 3 blocks of 4 elements each, each block is assigned to a processor, and each element in a block is assigned to some memory location.

  In general, a *partition* is a reshape morphism $g : D \to \sum(i{:}I)\, D(i)$, where I is some index domain and $\sum$ denotes generalized sum over all the $D(i)$'s. The idea is that the domain D is partitioned into disjoint union of subdomains $D(i)$, which may contain different number of elements.

There are numerous other forms of reshape morphisms ranging from "piece-wise affine" morphisms for more complex loop transformations [?], to those that are mutually recursive with the program (to be transformed) for dynamic data distribution.

Refinement Morphisms A domain morphism that does not have an inverse is called *refinement morphism*. Though we shall not deal with them in this paper, refinement morphisms are useful in representing more complex relationships between domains, such as the transformation of one-to-many broadcasts into binary tree broadcasting [?].

2.4 Data Fields and Data Field Morphisms

Definition A *data field* is a function over some index domain D into some domain of values V.

Usually, V will be the integers or the floating point numbers; however, for higher-order data fields, it can be some domain of data fields. Data fields unify the notion of distributed data structures, such as arrays, and functions. A parallel computation is specified by a set of data field definitions, which may be mutually recursive.

To illustrate the use of data fields, consider the following program segment written in some imperative language (assuming there are no other statements assigning values to A):

```
float array A(0..n,0..n);
if i=0 or j=0 then A:=e1;
for i:= 2 to n do {
   for j := 2, n do {
      A(i,j) := A(i-1,j) + A(i,j-1) } }
```

Let V be the data type of floating point numbers, in the notation of data fields, the above is written as

$$D_0 = \mathsf{interval}(0,n) \qquad D = D_0 \times D_0$$

$$a{:}D \to V = \mathsf{fn}(i,j)\left\{\begin{array}{l} i = 0 \vee j = 0 \to e_1 \\ \mathsf{else} \to a(i-1,j) + a(i,j-1) \end{array}\right\}$$

The primary role of index of domain morphisms is in defining new data fields.

Definition A *data field morphism* induced by an index domain morphism $g : D \to E$, is a mapping

$$g^* : (E \to V) \to (D \to V) : a \mapsto a \circ g$$

where $D \to V$ and $E \to V$ are sets of data fields.

Given a data field $a : D \to V$ and a domain morphism $g : D \to E$, what we generally want is to find the new data field $\hat{a}$ such that $g^*(\hat{a}) = a$. In order to solve this equation we need the inverse of g, i.e. g needs to be a reshape morphism. Then given g and g^{-1}, we can formally derive $\hat{a} = g^{-1*}(a) = a \circ g^{-1}$.

2.5 Constructs for other Semantic Entities

Since a compiler relies on syntactic cues to do its optimizations, we found it important to make explicit certain semantic entities which are implicit in current programming languages, hyper-surfaces and communication forms.

Hyper-Surfaces Hyper-surfaces define the boundaries of domains where functions take on special values. The idea is to elevate boundaries of domains to be semantic entities and promote a programming style where the same semantic entity has a single syntactic entity corresponding to it. This way, the repetitive occurrence of related boolean expressions for testing a particular boundary often seen in programs can be eliminated. The notion of hyper-surface makes the algebraic program transformation easier and more elegant, as well as helps the compiler recognize domain boundaries and do optimizations such as aligning multiple data structures and reducing storage use.

In this paper, the notion of hyper-surfaces is used only in the example of Section ??, we therefore give the definitions in Section ?? of the Appendix.

Communication Forms Given an index domain D, a *communication form*, γ, is a composition of communication operators.[1] The idea is that given an index (considered as the receiver of a message), the communication form produces the "sending" index of the message. Conversely, the left inverse, if it exists,[2] of a communication form indicates the "receiving" index given an index interpreted as the sender of a message.

An asynchronous, distributed-memory parallel machine, at its lowest level of communication system, requires to know both the sender and receiver in order to establish synchronization and communication.[3] For the compiler of any parallel language without explicit "send" and "receive" commands, communication forms must be extracted and its inverse derived or provided by user in the form of directives.

In the following, we show the effect of domain morphisms on communication forms, which provides vital information for generating communication commands in the target code.

2.6 Effect of Reshape Morphism on Communication Form

Consider a data field $a : D \to V$ defined by

$$a = \mathsf{fn}(i){:}D\{\, \tau[\, b(\gamma(i))\,]\,\}$$

which contains a call to data field $b : D \to V$ using the communication form γ on index i. Let $g : D \to E[\mathcal{P} \times \mathcal{M}]$ be a reshape morphism from D to E with inverse g^{-1}. The resulting data fields after the reshape transformation are $\hat{a} = a \circ g^{-1}$ and $\hat{b} = b \circ g^{-1}$. Suppose γ is bijective and γ^{-1} given,[4] then the new communication form and its inverse after the transformation are

$$\hat{\gamma} = g \circ \gamma \circ g^{-1} \quad \text{and} \quad \hat{\gamma}^{-1} = g \circ \gamma^{-1} \circ g^{-1}$$

as shown in the commuting diagram of Figure ??.

The formulation of the new communication form shows that in compiler design and parallel language design, we should do the following: (1) By all means, try to obtain closed forms for γ and its inverse at compile time so that they do not have to be evaluated at runtime. (2)

[1] This is only a special type known as a *simple communication form*. In general, a communication form can be defined to be a set of relations representing multiple many-to-many communications.

[2] Again, in general, we consider the inverse relation, which always exists.

[3] Please refer to Hoare's CSP model [?].

[4] Recall that, in general, γ is a set of relations and the result generalizes accordingly.

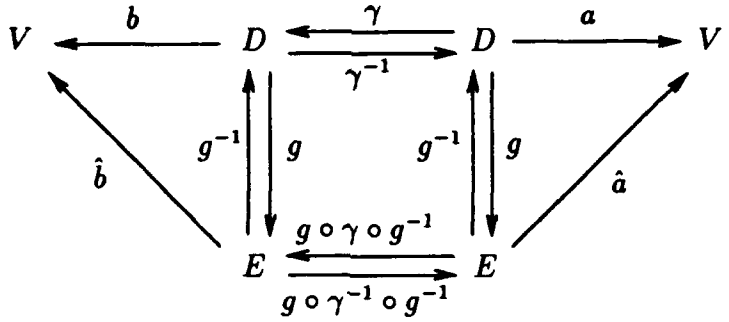

Figure 1: Diagram showing the new communication form induced by domain morphism g and communication form γ.

If the above is not possible, devise an efficient run-time implementation to evaluate γ and its inverse. (3) The information needed to do the above is contained in the original communication forms (and their inverses) and the domain morphism (and its inverse); hence, the compiler must generate the morphisms and the communication forms either automatically or rely on user provided directives.

3 Program Transformation

Data fields and domain morphisms are semantic entities which are represented in a programming language. Semantically, a new data field can be defined as the composition of a data field and a domain morphism. In this section we describe an equational theory that allows the representation of the new data field to be obtained by manipulating the representations of the given data field and domain morphism. First, we introduce the equational theory, and then describe the steps in the program transformation using reshape morphisms.

3.1 Equational Theory

By an equational theory of a programming language we mean a set of valid equations or algebraic identities in the language ($M = N$) along with inference rules for deriving new equations from old. It is based on the equational theory of Church's lambda calculus [?]. The α and β rules are well understood, but it turns out that η-abstraction is essential in order to simplify certain expressions. Figure ?? lists a minimal set of algebraic identities for a functional language with function-abstraction, the conditional expression and composition of functions. Some of these identities are listed in Backus's work on FP [?], though the key difference is that he does not have any involving explicit function-abstraction.

Figure ?? lists the basic inference rules including the usual ones for equality (reflexivity, symmetry, and transitivity), substitution, as well as ones obtained from application, abstraction and composition.

A definition, in an equational theory, is a simple equation where the left hand side is a single variable and the right hand side is some expression which may contain the same variable (recursive definitions). In the framework of [?], a definition enriches a theory with a new operator and a new equation. For a mutually recursive set of function definitions, we

$$\begin{aligned} \mathsf{fn}(x){:}T\{M\} &= \mathsf{fn}(y){:}T\{M[x/y]\} && (\alpha)\\ \mathsf{fn}(x){:}T\{M\}(N) &= M[x/N] && (\beta)\\ \mathsf{fn}(x){:}T\{M(x)\} &= M \quad (x \text{ not free in } M) && (\eta) \end{aligned}$$

Figure 2: Algebraic identities from the λ-calculus, where M and N are expressions and T is a type-expression.

$$\begin{aligned} F \circ \{B \to H\} &= \{B \to F \circ H\} && \text{(l-dist-comp-if)}\\ \{B \to H\} \circ F &= \{B \circ F \to H \circ F\} && \text{(r-dist-comp-if)}\\ \{B \to H\}(x) &= \{B(x) \to H(x)\} && \text{(dist-app-if)}\\ \mathsf{fn}(x)\{B(x) \to H(x)\} &= \{\mathsf{fn}(x)\{B(x)\} \to \mathsf{fn}(x)\{H(x)\}\} && \text{(dist-abs-if)} \end{aligned}$$

Figure 3: Algebraic identities involving the conditional, where F and H are function expressions, and B is a boolean function expression.

$$M = M \qquad \frac{M = N}{N = M} \qquad \frac{M = L \text{ and } L = N}{M = N}$$

$$\frac{M = N}{M \circ F = N \circ F} \qquad \frac{M = N}{H \circ M = H \circ N}$$

$$\frac{M = N}{\mathsf{fn}(x)\{M\} = \mathsf{fn}(x)\{N\}} \qquad \frac{M = N}{M(L) = N(L)} \qquad \frac{M = N}{L(M) = L(N)}$$

$$\frac{M = N}{M[H/K] = N[H/K]}$$

Figure 4: Inference rules, where $M[H/K]$ denotes the new term with H replaced by K, the substitution operation of the λ-calculus.

consider the functions that are implicitly defined as satisfying all the defining equations in the enriched theory.

3.2 Strategy for Obtaining New Definitions

With the equational theory providing the algebraic identities and the inference rules, we describe the strategy of formally transform the original program into a more efficient one.

For simplicity we begin with a program consisting of one definition:

$$a = \mathsf{fn}(x){:}D\{\tau_1[a]\},$$

where $\tau_1[a]$ is an expression in x possibly containing a. By an abuse of notation, we also use a to denote the data field defined. Next, let the reshape morphism g and its inverse be given by

$$g = \mathsf{fn}(x){:}D\{\tau_2{:}E\} \quad \text{and} \quad g^{-1} = \mathsf{fn}(y){:}E\{\tau_3{:}D\}.$$

Semantically, what we want is a data field $\hat{a}$ satisfying $\hat{a} = a \circ g^{-1}$. However, merely executing the program g^{-1} followed by a does not decrease the communication cost. What we want is

a new definition of $\hat{a}$ which does not contain either a, g or g^{-1}. A strategy for obtaining a new definition for $\hat{a}$ from the definitions of a, g and g^{-1} is the following:

1. Using the identity $a = \hat{a} \circ g$, replace all occurrences of a with $\hat{a} \circ g$ in the definition of a.

2. Using a combination of unfoldings of g and g^{-1} and various other identities given in the theory, eliminate all occurrences of g and g^{-1} from the result of the first step. A very useful transformation turns out to be the η-abstraction, where we provide a function with dummy arguments in order to unfold it.

3.3 Metalanguage

Since the equational theory deals with programs, and equations, a metalanguage for manipulating program expressions can be defined [?]. The derivation strategy described above can be programmed in the metalanguage using metalanguage operators, the constructors and selectors.

In the following example derivation, the metalanguage operators that achieve the transformation appear in square brackets.

4 Example: Successive Over-Relaxation

We illustrate our program transformation strategy on the successive over-relaxation defined in **Program-1**. This program solves Laplace's equation using finite difference formulation. The program finds solutions for the unknown potential within a finite, discretized 2-dimensional spatial domain (defined by index domain $D_0 \times D_0$) with fixed boundary values and initial values (conditional branches guarded by hyper-surfaces S_0 and S_1, respectively). The algorithm is iterative (over index domain D_1), and in each iteration the potential at every point of the 2-dimensional grid is the average of the values of its four neighboring points and itself. The over-relaxation ordering is such that the values of the current iteration are used for the upper $((i-1,j))$ and left $((i-1,j))$ neighbors, while those from the previous iterations are used for itself $((i,j))$ and the other two neighbors. This algorithm uses $O(n^2)$ space but only $O(n)$ parallelism due to the over-relaxation ordering. The point of the domain morphism is to transform this program to another that has $O(n^2)$ parallelism.

Program-1

! Index Domains

$$D_0 = \mathsf{interval}(0,n) \qquad D_1 = \mathsf{interval}(1,m) \qquad D = \mathsf{prod\text{-}dom}(D_0, D_0, D_1)$$

! Hyper-Surfaces

$$S_0 = \mathsf{fn}(i,j,k){:}D\{(i=0,>) \wedge (i=n,<) \wedge (j=0,>) \wedge (j=n,<)\}$$
$$S_1 = \mathsf{fn}(i,j,k){:}D\{(k=1,>) \wedge (k=m,<)\}$$

! Data Field

$$a = \mathsf{fn}(i,j,k){:}D$$
$$\left\{ \begin{array}{l} S_0^{\circ}(i,j,k) \rightarrow 0 \\ \\ S_0^{+}(i,j,k) \rightarrow \left\{ \begin{array}{l} S_1^{\circ}(i,j,k) \rightarrow A_0(i,j) \\ S_1^{+}(i,j,k) \rightarrow f(a(i-1,j,k), a(i,j-1,k), \\ \qquad\qquad a(i,j,k-1), a(i+1,j,k-1), \\ \qquad\qquad a(i,j+1,k-1)) \end{array} \right\} \end{array} \right\}$$

Although the program uses hyper-surfaces in the guards (see Section ?? for detailed descriptions), for comparison, we present the equivalent boolean expressions:

$$S_0^{\circ}(i,j,k) \equiv (i=0 \vee i=n \vee j=0 \vee j=n) \wedge (0 \le i \le n) \wedge (0 \le j \le n) \wedge (1 \le k \le m)$$
$$S_0^{+}(i,j,k) \equiv (0 < i < n) \wedge (0 < j < n) \wedge (1 \le k \le m)$$
$$S_1^{\circ}(i,j,k) \equiv (k=1 \vee k=m) \wedge (0 \le i \le n) \wedge (0 \le j \le n) \wedge (1 \le k \le m)$$
$$S_1^{+}(i,j,k) \equiv (0 \le i \le n) \wedge (0 \le j \le n) \wedge (1 < k < m)$$

The use of hyper-surfaces not only allows the compiler to extract useful optimization information from the program, but also provides conceptual tools for reasoning about the different boundaries.

4.1 New Domains and Domain Morphisms

First, we define the new domain E and domain morphisms $g : D \rightarrow E$ and $g^{-1} : E \rightarrow D$. The kind of domain E indicates that the first two coordinates are to be interpreted as spatial, and the third temporal.

Morphisms

$$E = \mathsf{image}(g', D)\ \mathsf{where}\{\ g' = \mathsf{fn}(i,j,k){:}D\{(i,j,i+j+2*k)\}\ \}$$
$$g = \mathsf{fn}(i,j,k){:}D\{(i,j,i+j+2*k){:}E[\mathcal{S}\times\mathcal{S}\times\mathcal{T}]\}$$
$$g^{-1} = \mathsf{fn}(i,j,k){:}E\{(i,j,(k-i-j)/2){:}D\}.$$

Next, define new surfaces R_0 and R_1 which satisfy:

$$R_0 = S_0 \circ g^{-1} \qquad R_1 = S_1 \circ g^{-1}$$
$$S_0 = R_0 \circ g \qquad S_1 = R_1 \circ g$$

4.2 Hyper-Surface Derivations

$$\begin{aligned}
R_1 &= S_1 \circ g^{-1} \\
&= \mathsf{fn}(i,j,k){:}E\{S_1 \circ g^{-1}(i,j,k)\} \\
&= \mathsf{fn}(i,j,k){:}E\{S_1(i,j,(k-i-j)/2)\} \qquad \text{(Unfold composition and } g^{-1}\text{)} \\
&= \mathsf{fn}(i,j,k){:}E\{\mathsf{fn}(i,j,k){:}D\{(k=1,>)\wedge(k=m,<)\}(i,j,(k-i-j)/2)\} \\
&= \mathsf{fn}(i,j,k){:}E\{((k-i-j)/2=1,>)\wedge((k-i-j)/2=m,<)\} \\
&= \mathsf{fn}(i,j,k){:}E\{(k-i-j=2,>)\wedge(k-i-j=2*m,<))\}
\end{aligned}$$

Similarly, we obtain

$$R_0 = \mathsf{fn}(i,j,k){:}E\{(i=0,>)\wedge(i=n,<)\wedge(j=0,>)\wedge(j=n,<)\}$$

which has the same body as S_0 since there are no constraints on the third coordinate (k).

4.3 Data Field Derivation

Let κ_0 denote the equation defining a in **Program-1**. The derivation of the definition for the new data field $\hat{a}$ makes use of the identities

$$\hat{a} = a \circ g^{-1} \quad \text{and} \quad a = \hat{a} \circ g.$$

(Note: we adopt the convention that composition binds more tightly than application, so that $f \circ g(x) = f(g(x))$.)

1. Substitute $\ulcorner a \urcorner$ with $\ulcorner \hat{a} \circ g \urcorner$ in κ_0 $[\kappa_1 = \mathsf{subst}(\kappa_0, \ulcorner a \urcorner, \ulcorner \hat{a} \circ g \urcorner)]$:

$$\begin{aligned}
\hat{a} \circ g \;=\; & \mathsf{fn}(i,j,k){:}D \\
& \left\{ \begin{array}{l}
S_0^\circ(i,j,k) \rightarrow 0 \\
S_0^+(i,j,k) \rightarrow \left\{ \begin{array}{l}
S_1^\circ(i,j,k) \rightarrow A_0(i,j) \\
S_1^+(i,j,k) \rightarrow f(\hat{a} \circ g(i-1,j,k), \hat{a} \circ g(i,j-1,k), \\
\qquad\qquad \hat{a} \circ g(i,j,k-1), \hat{a} \circ g(i+1,j,k-1), \\
\qquad\qquad \hat{a} \circ g(i,j+1,k-1))
\end{array} \right.
\end{array} \right\}
\end{aligned}$$

2. Right-compose both sides of κ_1 with $\ulcorner g^{-1} \urcorner$, and then simplify compositions by eliminating identities ($g \circ g^{-1} \triangleright 1_E$ and $g^{-1} \circ g \triangleright 1_D$) and unfold the composition operator when possible ($f \circ g(x) \triangleright f(g(x))$)
$[\kappa_2 = \mathsf{simplify\text{-}comp}(\mathsf{right\text{-}comp}(\kappa_1, \ulcorner g^{-1} \urcorner))]$:

$$\begin{aligned}
\hat{a} \;=\; & [\mathsf{fn}(i,j,k){:}D \\
& \left\{ \begin{array}{l}
S_0^\circ(i,j,k) \rightarrow 0 \\
S_0^+(i,j,k) \rightarrow \left\{ \begin{array}{l}
S_1^\circ(i,j,k) \rightarrow A_0(i,j) \\
S_1^+(i,j,k) \rightarrow f(\hat{a}(g(i-1,j,k)), \hat{a}(g(i,j-1,k)), \\
\qquad\qquad \hat{a}(g(i,j,k-1)), \hat{a}(g(i+1,j,k-1)), \\
\qquad\qquad \hat{a}(g(i,j+1,k-1)))
\end{array} \right.
\end{array} \right\}] \circ g^{-1}
\end{aligned}$$

3. Distribute the abstraction $\ulcorner(i,j,k)\!:\!D\urcorner$ over the conditional $[\kappa_3 = \textsf{dist-abs-if}(\kappa_2, \ulcorner(i,j,k)\!:\!D\urcorner)]$:

$$\hat{a} = \left\{ \begin{array}{l} \textsf{fn}(i,j,k)\!:\!D\{S_0^\circ(i,j,k)\} \to \textsf{fn}(i,j,k)\!:\!D\{0\} \\ \textsf{fn}(i,j,k)\!:\!D\{S_0^+(i,j,k)\} \to \\ \quad \left\{ \begin{array}{l} \textsf{fn}(i,j,k)\!:\!D\{S_1^\circ(i,j,k)\} \to \textsf{fn}(i,j,k)\!:\!D\{A_0(i,j)\} \\ \textsf{fn}(i,j,k)\!:\!D\{S_1^+(i,j,k)\} \to \\ \quad \textsf{fn}(i,j,k)\!:\!D\{ \\ \qquad f(\hat{a}(g(i-1,j,k)), \hat{a}(g(i,j-1,k)), \\ \qquad\quad \hat{a}(g(i,j,k-1)), \hat{a}(g(i+1,j,k-1)), \\ \qquad\quad \hat{a}(g(i,j+1,k-1)))\} \end{array} \right\} \end{array} \right\} \circ g^{-1}$$

4. Perform η-reductions $(\textsf{fn}(x)\!:\!D\{M(x)\} \triangleright M)$ $[\kappa_4 = \textsf{eta-reduce}(\kappa_3)]$:

$$\hat{a} = \left\{ \begin{array}{l} S_0^\circ \to \textsf{fn}(i,j,k)\!:\!D\{0\} \\ S_0^+ \to \left\{ \begin{array}{l} S_1^\circ \to \textsf{fn}(i,j,k)\!:\!D\{A_0(i,j)\} \\ S_1^+ \to \textsf{fn}(i,j,k)\!:\!D\{ \\ \quad f(\hat{a}(g(i-1,j,k)), \hat{a}(g(i,j-1,k)), \\ \qquad \hat{a}(g(i,j,k-1)), \hat{a}(g(i+1,j,k-1)), \\ \qquad \hat{a}(g(i,j+1,k-1)))\} \end{array} \right\} \end{array} \right\} \circ g^{-1}$$

5. Right-distribute the composition over the conditional $[\kappa_5 = \textsf{r-dist-comp-if}(\kappa_4)]$:

$$\hat{a} = \left\{ \begin{array}{l} S_0^\circ \circ g^{-1} \to (\textsf{fn}(i,j,k)\!:\!D\{0\}) \circ g^{-1} \\ S_0^+ \circ g^{-1} \to \left\{ \begin{array}{l} S_1^\circ \circ g^{-1} \to (\textsf{fn}(i,j,k)\!:\!D\{A_0(i,j)\}) \circ g^{-1} \\ S_1^+ \circ g^{-1} \to (\textsf{fn}(i,j,k)\!:\!D\{ \\ \quad f(\hat{a}(g(i-1,j,k)), \hat{a}(g(i,j-1,k)), \\ \qquad \hat{a}(g(i,j,k-1)), \hat{a}(g(i+1,j,k-1)), \\ \qquad \hat{a}(g(i,j+1,k-1)))\}) \circ g^{-1} \end{array} \right\} \end{array} \right\}$$

6. Substitute $\ulcorner S_0^\circ \circ g^{-1}\urcorner$ with $\ulcorner R_0^\circ\urcorner$ and $\ulcorner S_1^\circ \circ g^{-1}\urcorner$ with $\ulcorner R_1^\circ\urcorner$ $[\kappa_6 = \textsf{subst}(\textsf{subst}(\kappa_5, \ulcorner S_0^\circ \circ g^{-1}\urcorner, \ulcorner R_0^\circ\urcorner), \ulcorner S_1^\circ \circ g^{-1}\urcorner, \ulcorner R_1^\circ\urcorner)]$:

$$\hat{a} = \left\{ \begin{array}{l} R_0^\circ \to (\textsf{fn}(i,j,k)\!:\!D\{0\}) \circ g^{-1} \\ R_0^+ \to \left\{ \begin{array}{l} R_1^\circ \to (\textsf{fn}(i,j,k)\!:\!D\{A_0(i,j)\}) \circ g^{-1} \\ R_1^+ \to (\textsf{fn}(i,j,k)\!:\!D\{ \\ \quad f(\hat{a}(g(i-1,j,k)), \hat{a}(g(i,j-1,k)), \\ \qquad \hat{a}(g(i,j,k-1)), \hat{a}(g(i+1,j,k-1)), \\ \qquad \hat{a}(g(i,j+1,k-1)))\}) \circ g^{-1} \end{array} \right\} \end{array} \right\}$$

7. Eta-abstract the right-hand side of the equation by $\ulcorner(i,j,k){:}E\urcorner$ $[\kappa_7 = \mathsf{mk\text{-}eqn}(\ulcorner\hat{a}\urcorner, \mathsf{mk\text{-}eta}(\ulcorner(i,j,k){:}E\urcorner, \mathsf{rhs}(\kappa_6)))]$:

$$\hat{a} = \mathsf{fn}(i,j,k){:}E\{ \left\{ \begin{array}{l} R_0^\circ \to (\mathsf{fn}(i,j,k){:}D\{0\}) \circ g^{-1} \\ R_0^+ \to \left\{ \begin{array}{l} R_1^\circ \to (\mathsf{fn}(i,j,k){:}D\{A_0(i,j)\}) \circ g^{-1} \\ R_1^+ \to (\mathsf{fn}(i,j,k){:}D\{ \\ \quad f(\hat{a}(g(i-1,j,k)), \hat{a}(g(i,j-1,k)), \\ \quad\quad \hat{a}(g(i,j,k-1)), \hat{a}(g(i+1,j,k-1)), \\ \quad\quad \hat{a}(g(i,j+1,k-1)))\}) \circ g^{-1} \end{array} \right\} \end{array} \right\} (i,j,k)\}$$

8. Distribute application over conditional $[\kappa_8 = \mathsf{dist\text{-}app\text{-}if}(\kappa_7)]$:

$$\hat{a} = \mathsf{fn}(i,j,k){:}E\{ \left\{ \begin{array}{l} R_0^\circ(i,j,k) \to (\mathsf{fn}(i,j,k){:}D\{0\}) \circ g^{-1}(i,j,k) \\ R_0^+(i,j,k) \to \left\{ \begin{array}{l} R_1^\circ(i,j,k) \to (\mathsf{fn}(i,j,k){:}D\{A_0(i,j)\}) \circ g^{-1}(i,j,k) \\ R_1^+(i,j,k) \to \\ \quad (\mathsf{fn}(i,j,k){:}D\{ \\ \quad\quad f(\hat{a}(g(i-1,j,k)), \hat{a}(g(i,j-1,k)), \\ \quad\quad\quad \hat{a}(g(i,j,k-1)), \hat{a}(g(i+1,j,k-1)), \\ \quad\quad\quad \hat{a}(g(i,j+1,k-1)))\}) \circ g^{-1}(i,j,k) \end{array} \right\} \end{array} \right\}\}$$

9. Simplify composition $[\kappa_9 = \mathsf{simplify\text{-}comp}(\kappa_8)]$:

$$\hat{a} = \mathsf{fn}(i,j,k){:}E\{ \left\{ \begin{array}{l} R_0^\circ(i,j,k) \to (\mathsf{fn}(i,j,k){:}D\{0\})(g^{-1}(i,j,k)) \\ R_0^+(i,j,k) \to \left\{ \begin{array}{l} R_1^\circ(i,j,k) \to (\mathsf{fn}(i,j,k){:}D\{A_0(i,j)\})(g^{-1}(i,j,k)) \\ R_1^+(i,j,k) \to \\ \quad (\mathsf{fn}(i,j,k){:}D\{ \\ \quad\quad f(\hat{a}(g(i-1,j,k)), \hat{a}(g(i,j-1,k)), \\ \quad\quad\quad \hat{a}(g(i,j,k-1)), \hat{a}(g(i+1,j,k-1)), \\ \quad\quad\quad \hat{a}(g(i,j+1,k-1)))\})(g^{-1}(i,j,k)) \end{array} \right\} \end{array} \right\}\}$$

10. Unfold $\ulcorner g^{-1}\urcorner$ and $\ulcorner g\urcorner$ $[\kappa_{10} = \mathsf{unfold}(\mathsf{unfold}(\kappa_9, \ulcorner g^{-1}\urcorner), \ulcorner g\urcorner)]$:

$$\hat{a} = \mathsf{fn}(i,j,k){:}E\{$$

$$\left\{\begin{array}{l} R_0^\circ(i,j,k) \rightarrow (\mathsf{fn}(i,j,k){:}D\{0\})(i,j,(k-i-j)/2) \\ R_0^+(i,j,k) \rightarrow \\ \quad \left\{\begin{array}{l} R_1^\circ(i,j,k) \rightarrow (\mathsf{fn}(i,j,k){:}D\{A_0(i,j)\})(i,j,(k-i-j)/2) \\ R_1^+(i,j,k) \rightarrow \\ \quad (\mathsf{fn}(i,j,k){:}D\{ \\ \quad\quad f(\hat{a}(i-1,j,i-1+j+2*k),\hat{a}(i,j-1,i+j-1+2*k), \\ \quad\quad\quad \hat{a}(i,j,i+j+2*k-2),\hat{a}(i+1,j,i+j+2*k-1), \\ \quad\quad\quad \hat{a}(i,j+1,i+j+2*k-1))\})(i,j,(k-i-j)/2) \end{array}\right\} \end{array}\right\}$$

11. Perform β-reduction and simplify the arithmetic $[\kappa_{11} = \mathsf{simlify\text{-}arith}(\mathsf{reduce}(\kappa_{10}))]$:

$$\hat{a} = \mathsf{fn}(i,j,k){:}E\{ \left\{\begin{array}{l} R_0^\circ(i,j,k) \rightarrow 0 \\ R_0^+(i,j,k) \rightarrow \left\{\begin{array}{l} R_1^\circ(i,j,k) \rightarrow A_0(i,j) \\ R_1^+(i,j,k) \rightarrow f(\hat{a}(i-1,j,k-1),\hat{a}(i,j-1,k-1), \\ \qquad\qquad \hat{a}(i,j,k-2),\hat{a}(i+1,j,k-1), \\ \qquad\qquad \hat{a}(i,j+1,k-1)) \end{array}\right\} \end{array}\right\}\}$$

The resulting index domain and data field definitions appear in **Program-2**. The program captures the red/black checkerboard algorithm of SOR described in Chapter 7 of [?], except for the final partitioning of the two spatial coordinates onto processors and memory, which again can be represented as a domain morphism and the transformation performed with the same procedure but with difference in the detail in arithmetic simplification.

Program-2

! Index Domain

$$E = \mathsf{image}(g', D)\ \mathsf{where}\{\ g' = \mathsf{fn}(i,j,k){:}D\{(i,j,i+j+2*k)\}\ \}$$

! Hyper-Surfaces

$$R_0 = \mathsf{fn}(i,j,k){:}E\{(i=0,>) \wedge (i=n,<) \wedge (j=0,>) \wedge (j=n,<)\}$$

$$R_1 = \mathsf{fn}(i,j,k){:}E\{(k-i-j=2,>) \wedge (k-i-j=2*m,<))\}$$

! Data Field

$$\hat{a} = \mathsf{fn}(i,j,k){:}E \left\{\begin{array}{l} R_0^\circ(i,j,k) \rightarrow 0 \\ R_0^+(i,j,k) \rightarrow \left\{\begin{array}{l} R_1^\circ(i,j,k) \rightarrow A_0(i,j) \\ R_1^+(i,j,k) \rightarrow f(\hat{a}(i-1,j,k-1),\hat{a}(i,j-1,k-1), \\ \qquad\qquad \hat{a}(i,j,k-2),\hat{a}(i+1,j,k-1), \\ \qquad\qquad \hat{a}(i,j+1,k-1)) \end{array}\right\} \end{array}\right\}$$

We translate **Program-2** into an imperative program to illustrate the effect of the domain morphism on the implementation. We begin with the data structure. By analyzing data

dependence of $\hat{a}$, we know that a two dimensional array is needed to store the values of $\hat{a}$ at each point of the spatial component of domain E. The recursive call made to $a(i,j,k-2)$ indicates a value of 2 time step ago is needed. Since each element of a is changed only once every two time steps, there is no additional storage needed.

Next, the control structure. The bounds of the `for`-loop is derived from the temporal component (indexed by k) of domain E, while those of the `forall`-loop from the spatial component of E. The red/black checkerboard ordering of the over-relaxation is controlled by the predicates of the `if`-statement in the `forall`-loop. These predicates are derived from the hyper-surface R_1 and the domain E. In particular, the test `whole?((k-i-j)/2)` is to ensure only those elements of E that are images of elements of D (which are integral) participate in the computation. Note that because of the limitation of the syntax for specifying loop bounds in current programming languages, the domain information must be specified both in the loop bounds and in the `if`-statement.

Note that **Program-2** contains all the necessary information for a translator to generate the imperative program.

```
function SOR(A0)
  float array A(0..n,0..n);
  A(i,j) := A0(i,j);
  for k:= 2 to 2*(n+m) do {
   forall (i,j) in (0..n , 0..n) do {
    if (2*m > k-i-j) and (k-i-j > 2) and whole?((k-i-j)/2)
    then A(i,j) := f(A(i-1,j), A(i,j-1), A(i,j), A(i+1,j), A(i,j+1))
    fi } }
```

5 Concluding Remarks

The complexity of managing explicit parallelism to produce efficient code remains a major obstacle in the widespread use of the new generation of parallel machines. At the same time, the task of automatically generating parallel programs appears to be extremely difficult and costly, except for a very restricted class of problems. Recognizing these issues, we know that a balance must be sought between the automatable and the effort required of the programmer.

Our work provides a theoretical basis for and makes evident where to draw the line between what is automatable and what needs user-directives. As far as we know, this is a first attempt at formalizing parallelizing compiler techniques. One of its consequence is to provide a framework for systematic application of these techniques, where a *reference metric* is defined over the communication forms to provide guidance in minimizing certain cost. Parallelization and optimization techniques have, by and large, been done in an ad hoc fashion, and we have just began to understand their interactions now that this theoretical framework is in place.

A Hyper-Surfaces

Hyper-surfaces define the boundaries of domains where functions take on special values. By making hyper-surfaces first class objects, we eliminate the repetitive occurrence of related boolean expressions for boundary testing which makes compiler optimization difficult.

A data field definition will usually contain a conditional expression which tests for certain boundary conditions from the interior of the domain of definition. These tests are usually expressed as inequalities over the formal parameters. When many data fields are defined over the same domain, the tests may be combined and transformed into very different forms, making the domain analysis very difficult.

In order to preserve the semantic content of tests for boundary conditions, we introduce a general notion of a *hyper-surface* which specifies the boundary of a domain, and a number of operators for testing whether a point is on the positive or the negative "side" of the boundary, or on the boundary.

In the following, all manifolds are assumed to be simply connected, such as Euclidean spaces. Note, however, that the notion of hyper-surface can be extended to trees and other discrete structures which contain a notion of locality and can be partitioned into the positive and the negative components.

Definition For any n-dimensional manifold M, a *hyper-surface* is an $(n-1)$-dimensional sub-manifold S which partitions M into two components known as the *positive* (S^+_M) and the *negative* (S^-_M) components.

For each surface S in M, we define three associated boolean valued functions. For each point x in M:

$$\begin{array}{ll} S^\circ(x) \text{ iff } x \in S & \text{(surface)} \\ S^+(x) \text{ iff } x \in S^+_M & \text{(positive-component)} \\ S^-(x) \text{ iff } x \in S^-_M & \text{(negative-component)} \end{array}$$

By an abuse of notation, we write $S(x)$ for $S^\circ(x)$. Usually, we just write S^+ when the ambient manifold M is understood.

A.1 Representation

For n-dimensional Euclidean space over the reals, a hyper-surface can be specified by an equation $\tau[x_1, \ldots, x_n] = 0$, where the left hand side contains at least one index x_i with non-zero coefficient, and an orientation. One way to specify the positive component with respect to the hyper-surface is by modifying the equation into an inequation. There are two possibilities, the positive component can be defined as the set of points which satisfy either $\tau < 0$ or $\tau > 0$. We formalize this into a hyper-surface constructor.

Definition Let D be an n-dimensional manifold. The expression

$$\mathsf{fn}(x_1, \ldots, x_n) : D\{(\tau[x_1, \ldots, x_n] = 0, \gamma)\}$$

denotes a simple hyper-surface defined by the equation $\tau[x_1, \ldots, x_n] = 0$ and the positive component is given by γ, which can be either $<$ or $>$.

Example Let D be the 3-dimensional Euclidean space. Then

$$\mathsf{fn}(i,j,k) : D\{(k-(i+j)/2=0,>)\}$$

denotes a hyper-surface of D such that $S^{\circ}(i,j,k)$ iff $k-(i+j)/2=0$ and $S^{+}(i,j,k)$ iff $k-(i+j)/2>0$.

A.2 Hyper-Surface Operations

Let S_1 and S_2 be hyper-surfaces of M. We can define the *conjunction* $(S_1 \wedge S_2)$, *negation* $(\overline{S_1})$, *disjunction* $(S_1 \vee S_2)$ using the notion of positive component:

$$(S_1 \wedge S_2)^{+} = S_1^{+} \cap S_2^{+} \quad \text{and} \quad \overline{S}^{+} = S^{-} \quad \text{and} \quad S_1 \vee S_2 = \overline{\overline{S_1} \wedge \overline{S_2}}.$$

The effect of conjunction is to take the conjunction of the two defining equations, while that of negation is to switch $<$ and $>$.

In general, we can define complex hyper-surfaces by allowing logical conjunction, disjunction, and negation in the body of the hyper-surface constructor:

$$\begin{aligned}&\mathsf{fn}(x) : D\{(\tau_1[x]=0,\gamma_1) \wedge (\tau_2[x]=0,\gamma_2)\}\\ &= (\mathsf{fn}(x) : D\{(\tau_1[x]=0,\gamma_1)\}) \wedge (\mathsf{fn}(x) : D\{(\tau_2[x]=0,\gamma_2)\})\end{aligned}$$

$$\begin{aligned}&\mathsf{fn}(x) : D\{(\tau_1[x]=0,\gamma_1) \vee (\tau_2[x]=0,\gamma_2)\}\\ &= (\mathsf{fn}(x) : D\{(\tau_1[x]=0,\gamma_1)\}) \vee (\mathsf{fn}(x) : D\{(\tau_2[x]=0,\gamma_2)\})\end{aligned}$$

$$\mathsf{fn}(x) : D\{\neg(\tau[x]=0,\gamma)\} = \overline{\mathsf{fn}(x) : D\{(\tau[x]=0,\gamma)\}} = \mathsf{fn}(x) : D\{(\tau[x]=0,\gamma')\}$$

where γ' is one of $\{<,>\}$ which is the opposite of γ.